ARMY RESEARCH LABORATORY

ENVELOPE: A New Approach to Estimating the Delivered Performance of High Performance Processors

by Daniel M. Pressel

ARL-TR-2671 February 2002

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-2671 **February 2002**

ENVELOPE: A New Approach to Estimating the Delivered Performance of High Performance Processors

Daniel M. Pressel
Computational and Information Sciences Directorate, ARL

Abstract

Simulating a computer run can be an excellent method for identifying performance bottlenecks and is especially valuable when discussing systems that do not yet exist. Traditional simulations collect a program trace and then have a simulator execute some subset of the trace one instruction at a time. Unfortunately, all of the standard variants of this technique are far too slow to use on jobs for high-end High Performance Computers and Supercomputers. We have developed an approach based primarily on an analysis of the memory access patterns and the number of floating point operations being executed that will estimate the performance of any run in a small fixed amount of time (e.g., a few seconds or less). Experience has shown that the results are nearly always within a factor of 2 of the measured results and frequently are within 15% or better of the measured results.

Acknowledgments

The author wishes to thank Csaba Zoltani and Dixie Hisley of the **ARL** and Punyam Satya-narayana of Raytheon for providing the necessary Perfex data. He would also like to thank the many researchers who have published performance results for the **NAS** and Linpack Benchmarks. Special thanks go to Csaba Zoltani, Dixie Hisley, Punyam Satya-narayana, John Levesque, Marek Behr, Steve Schraml, Allan Snavely, Sirpa Saarinen, and Shirley Moore for providing performance data used in the testing of these tools. Additional thanks are extended to Susan Sassaman of Business Plus Corp. (BPC) for editorial services on this report.

Note: Definitions for boldface text can be found in the Glossary.

INTENTIONALLY LEFT BLANK

Contents

List of Tables

INTENTIONALLY LEFT BLANK

1. Introduction

Simulating a computer run can be an excellent method for identifying performance bottlenecks. This can be especially valuable when discussing systems that do not yet exist. As such, it can augment benchmarking efforts by helping to explain, or even predict, results as opposed to simply reporting them. Unfortunately, traditional techniques in this field have suffered from three constraints:

(1) Describing a modern processor architecture with sufficient fidelity to ascertain the validity of the results is difficult.

(2) The simulations are so expensive to run that many jobs were simply considered to be prohibitively expensive to simulate. This can be an especially serious problem in the areas of high-end High Performance Computers and Supercomputers.

(3) Even when it was practical to run a simulation, it was frequently impractical to run it multiple times to quantify the benefits of various approaches to code tuning.

An example of this can be found in a recent posting to the comp.arch news group by a quote from Christopher Brian Colohan, a graduate student at Carnegie Mellon University:

> I am currently working on a project which involves detailed processor simulation of the **SPEC**Int2000 benchmarks. We are encountering the usual problem with simulations: they are taking too long, even when we use the provided "test" input. (The test input for bzip2 takes 41 seconds to run on our SGI Origin machine -- simulating more than a second or so of real **CPU** time on our simulator gets kind of painful. . .).

In response to this posting, John Mashey of SGI responded:

> I think people have tried to achieve this effect by taking slices of SPEC and other benchmarks in order to get, for example, reference streams to analyze alternate designs for memory hierarchies (i.e., this is different from changing the input). For example, for some codes, the performance on a few iterations would model the entire computation but, of course, this doesn't work for others [1].

Note: Definitions for boldface text can be found in the Glossary.

In a similar vein, Naraig Manjikian [2] has stated that when using the SimpleScalar tool set on a 333-**MHz** Sun Ultra/10 workstation, the simulation rate was approximately 300,000 instructions/second (compared to a peak rate of over 1 billion instructions/second when executing code directly on the hardware).

Considering the BT, CG, LU, and SP **NAS** (class B) benchmarks the floating point operations count range from 54 to 686 billion floating point operations for a single run. On a 300-MHz R12000-based Origin 2000 using a single processor, this translates into measured run times of 1,250–9,700 seconds. Clearly, if simulating a 41-second run is a problem, these industry standard benchmarks for high performance computing must be all but out of reach.

In response to this problem, we have developed an entirely different approach to simulating these runs. This approach is based on the time tested concept of "Back-of-the-Envelope" calculations. With this in mind, we have named our program "ENVELOPE." Rather than trying to simulate every aspect of the microarchitecture, this approach assumes that the computer architects know how to design a processor. In particular, if they claim that the peak floating point speed is 1 **GFLOPS**, then we take them at their word and use that number in the simulation. This also applies to various numbers involving memory, cache, and TLB latency and bandwidth. When combined with other system parameters and information about the code to be run on the system, an estimated performance is produced in a small fixed amount of time (e.g., 1 second). This run time is short enough to allow one to easily investigate the effects of turning various hardware features (e.g., prefetching) on or off and/or investigate various ways in which the code might be tuned.

2. Description of the Simulator

The current version of ENVELOPE prompts the user for input (this can come from a redirected file), writes its output to standard output, and also writes an annotated copy of the input to the file ENVELOPE.INPUTS. This later file can easily be edited and used as input in a future run (the annotations will be ignored by ENVELOPE). The output is broken into two categories—prompts for input and results. Every line that is considered to be a result begins with a pound sign (#) so that it can easily be searched when using GREP. Under the direction of Shirley Moore of the University of Tennessee at Knoxville, work is under way to produce a friendlier user interface using Java.

The first 18 lines of input describe the hardware in terms of peak characteristics (e.g., peak bandwidth between the outermost level of cache and the processor, in million bytes per second), minimum characteristics (e.g., memory latency in

nanoseconds), fundamental values (e.g., cache line size in bytes), and miscellaneous descriptions regarding the friendliness of the processor's design (e.g., describing the pipeline depth as short, moderate, deep, or very deep). This last set of values is used to estimate how much the peak speed of the processor should be discounted. The goal here is to somewhat level the playing field between designs that are blazingly fast but terribly unfriendly to program and those that made tradeoffs between peak speed and usability by keeping the number of operations in flight low and/or supporting features such as Out-of-Order execution and/or register renaming. In most cases, this heuristic makes little difference in the final result, since the performance of the memory system is frequently the limiting factor.

The remaining questions ask the user to describe the software and its memory access patterns. In theory, this information can be derived through inspection of the user's program. In practice, only the simplest of programs can be analyzed in this manner with sufficient precision. As an aid to this process, a tool has been developed, which will be described in more detail in sections 4 and 5. The main characteristic here is that the variable usage can be broken down into the following categories:

(1) Variables that spend a significant amount of time mapped to a register and therefore have a negligible number of loads or stores associated with them.

(2) Scratch arrays that have potentially been sized to fit in one of the levels of cache. An estimate of the size of the working set is supplied by the user. This can also be used to estimate the parameters for any working set that might exist. The assumption here is that there is a negligible cache miss rate associated with this working set, but that the flow of data between the cache and the processor still needs to be modeled.

(3) Blocked arrays or alternatively a second, presumably larger, working set. If this is used to model a blocked access pattern and if the working set is too large to fit into cache, a STRIDE-N access pattern is assumed. However, if the option of treating this as just another working set is used and if it fails to fit into cache, a STRIDE-1 access pattern is assumed. This flexibility allows us to model the behavior of a wide range of programs that benefit from the presence of a large cache and is particularly useful for programs with two or more distinct sizes of working sets.

(4) Arrays accessed with a STRIDE-N access pattern. In most cases, this access pattern will result in a high TLB miss rate. Almost all programs that we have looked at have at least some data that is accessed in this way; although for well tuned codes, less than 0.1% of all loads/stores will fall into this category. The LU NAS benchmark is an interesting exception to this rule. It can have 0.45–0.57% of all loads/stores falling into this category (depending on the version of the code being used). Fortunately, on many

systems, the resulting TLB misses seem to hit in cache, resulting in an acceptable level of performance. Unfortunately, it can be difficult to predict such behavior without a priori knowledge and/or experimental results to compare to. However, when such results are available, the hardware parameter for the TLB latency can be adjusted to a more appropriate value (e.g., subtract off the cost of a cache miss from the normally used cost of a TLB miss).

(5) Arrays accessed with a STRIDE-1 access pattern.

For categories 3–5, the amount of data reuse (cache and register levels combined) can be specified. This allows accurate modeling of programs that might not have a working set, or alternatively, the working set might be orders of magnitude larger than the cache. Even so, the program need not be restricted to using each data item in a cache line just once per cache miss. However, for some usage patterns, a usage factor in the range of 1 to 2 is exactly what will be seen. The ability to specify the amount of data reuse supports the widest possible range of programs without requiring hard coding in any assumptions.

ENVELOPE has been extensively tested with several numerically intensive programs using a variety of **RISC** and **CISC** processors. It is also designed to handle codes with few, if any, floating point operations. Furthermore, it should be able to model other types of architectures (e.g., vector processors), although no attempt has been made to date to exercise either of these capabilities.

Table 1 shows an example of an input file describing the Linpack 100 × 100 benchmark running on an IBM SP with 375-MHz Power 3 Thin nodes. The parameters in this file were derived using a detailed analysis of the source code, as well as taking into account common compiler optimizations.

Table 2 shows an example of an input file describing the NAS CG class B benchmark (**MPI**). Since this benchmark uses an unstructured grid, which inhibits prefetching, prefetching has been disabled. Again, the system being modeled is an IBM SP with 375-MHz Power 3 Thin nodes.

The predicted level of performance for the Linpack 100 on the IBM SP (as previously described) is 359 **MFLOPS**, while the measured level of performance is 426 MFLOPS. The predicted performance is within 19% of the measured performance. If one assumes that the benchmark was run on a dedicated node, then the processor could have used a larger percentage of the memory bandwidth. This allows the predicted performance to increase to between 481 and 559 MFLOPS (depending on the precise limitations of the processors memory interface), or within 11–31% of the measured performance [3].

For the CG benchmark, the predicted level of performance is 53 MFLOPS. The measured level of performance is 46 MFLOPS, or within 15% of the predicted level of performance [4]. Table 3 contains additional results.

3. The Equations

This section will discuss some of the equations used by ENVELOPE in estimating the performance of a program. The complete set of equations is quite lengthy and is beyond the scope of this paper. However, this discussion should be adequate in giving one a feel for how this program works.

3.1 Commonly Scalar Values

The time spent loading and storing commonly used values into the registers is computed using the following equations. It is important to note that it is assumed that these values can be found in one or more levels of cache and these operations will not result in any TLB misses. There is also an implied assumption that a processor cannot do more than one memory operation per opportunity to launch a multiply-add instruction. A few processors can, in fact, do better than that under certain circumstances. For most RISC and CISC processors, this is not a concern. However, it might be an important consideration if this code were to be used to model the performance of a Cray C90.

$$\text{Runtime} = \text{Runtime} + \frac{\text{P\#SV@} * \text{\#S}}{\text{UR\#SV@} * \text{UL2BANDWIDTH}}, \tag{1}$$

where # can either be L for LOAD or S for STORE and @ can either be DEDICATED* or GENERAL.†

P#SV@ refers to the percentage of the total amount of data that is either being loaded or stored (depending on what # is) of the specified class of values (as specified by @). #S is the total amount of data that is either loaded or stored. Therefore, the numerator refers to the amount of data being loaded or stored for this class of data. UR#SV@ refers to the amount of reuse at the register level for the data in question. The higher the level of reuse, the fewer the loads and stores that will actually be executed. In other words, for programs like Linpack, the code might indicate that a value will be loaded for each and every multiply-add instruction. However, a smart compiler might actually perform the load just once. In that case, the value for UR#SV@ would be large enough to make this part of the calculation irrelevant. Finally, UL2BANDWIDTH is the bandwidth

* The term DEDICATED implies that the values will only be used in calculations involving other commonly used values. A prime example of this is Horner's algorithm for evaluating polynomial equations.

† The term GENERAL implies that while the value will be "locked" into a register, it will be used in conjunction with other classes of variables. Therefore, the cost of the floating point or integer calculations that this value is involved with will be charged to the other classes of variables. An example of this type of variable might be PI.

between the processor and the outermost level of cache (frequently, the L2 cache).

For the DEDICATED variables, it is also necessary to calculate the time spent performing calculations involving these variables. This calculation is fairly straightforward. The only complicating factor is that since some, if not all, of these calculations can be paired up with load and store instructions on today's superscalar processors, one must avoid counting these cycles twice. This can be done by subtracting the time spent on the loads and stores for DEDICATED variables from the time spent computing with them. Since it is possible that the loads and stores will take longer, one needs to take the maximum of the difference and zero to avoid overcompensation.

3.2 Scratch Arrays

The next order of business is to account for the time spent dealing with small scratch arrays that will normally be "locked" into one or more level of cache. This is not assumed to be the case, but will generally be the case. Using the same notation and terminology as in subsection 3.1, the equations will be:

$$\text{Runtime} = \text{Runtime} + \frac{\text{P\#SA@} * \text{\#S}}{\text{UR\#SA@} * \text{UL2BANDWIDTH}}. \qquad (2)$$

Again, one must also take into account the time spent performing caluclations that only involve the scratch array and the commonly used values discussed in section 3.1.

A complicating factor is that one must also take into account the cache misses associated with these scratch arrays. ENVELOPE assumes that the number of TLB misses associated with the scratch arrays will be negligible. If the working set for the scratch arrays is larger than the size of the outermost level of cache, then ENVELOPE will assume a STRIDE-1 access pattern. Otherwise, ENVELOPE assumes that there is a sufficient level of data reuse at the cache level that the cost of batching up the cache can be ignored. Since this will normally be the case, we will not discuss the equations used any other time.

3.3 Blocked Memory Access Patterns

This portion of the code can either be used to estimate the cost of a code segment with a blocked memory access pattern or to estimate the cost of code segment involving scratch arrays with a larger working set. In the latter case, the estimated size of the working set should be negated and specified as the block

size.* The details associated with the handling of the case where the block size (or second working set size) is larger than the size of the cache will not be discussed. However, they are similar to the cases discussed in subsections 3.4 and 3.5.

With everything fitting into cache, one still has to be concerned with the cache misses associated with batching up the cache. Furthermore, unlike subsection 3.2, ENVELOPE does not assume that this cost can be ignored. In the case of a blocked access pattern, the level of reuse might be fairly modest (e.g., 2–10). Even in the case of a second level of working set, the level of reuse might be more limited. This part of the program is split into the following two cases:

(1) In the case where prefetching has been essentially disabled by setting the maximum number of outstanding prefetches/cache misses to 1, the latency of a cache miss will effectively determine the usable memory bandwidth.

(2) In the other case, the memory bandwidth will be the limiting factor.†‡

The first step is to determine the number of cache misses. This number is not actually used for the calculation of runtime and performance. It is however used to estimate the number of bus transactions. Calculating the time required to perform the cache misses is much more complicated. Some of the complicating factors are as follows:

---

* The negative value is used as a flag when the working set is larger than the cache. If the block size is positive, then ENVELOPE will estimate the cost of the loads and stores as though this code segment was using a STRIDE-N access pattern (the most expensive access pattern). However, if the block size is negative, then ENVELOPE estimates the cost of the loads and stores as though this code segment was using a STRIDE-1 access pattern. While this access pattern is more expensive than living out of cache, it is significantly less expensive than a STRIDE-N access pattern.

† Earlier in the program, the memory bandwidth was adjusted, when necessary, to handle the situation in which the supported level of prefetching was insufficient to fully utilize the complete memory bandwidth.

‡ ENVELOPE actually supports three separate values for the memory bandwidth:

(1) The bandwidth when only performing loads.

(2) The bandwidth when only performing stores.

(3) The bandwidth when performing a balanced mix of loads and stores.

For some systems, the three values will be the same. However, for the HP PA 8XXX series of processors, on a well balanced system, the third value will actually be the sum of the first two values. As a result of this, it is necessary to determine the amount of data being loaded and stored, pair the loads and stores at the mixed bandwidth rate, and then finish up with any nonpaired loads or stores. This assumes that the code pairs loads and stores as much as possible. This is not always the case, but is a good assumption for codes that call the **BLAS** routines, copy arrays, transpose arrays, and perform a variety of other common operations. However, for a program which copies data into a small buffer that is "locked" into cache, pounds on the buffer, and then writes the results out to a large global array, this can be a poor assumption. Fortunately, for many systems, this discussion is academic, since the three values are identical. In the remaining situations, it appears as though the maximum error is an overestimation of the performance (under estimation of the run time) by a factor of 2.

(1) Additional memory traffic resulting from the coherency protocols for cache lines that are stored to.

(2) The allocation of memory bandwidth to handle the portion of the loads and stores that can be overlapped with each other.

(3) Calculating the time required for those loads or stores that could not be overlapped with stores or loads respectively.

(4) The grouping of data into data structures. In particular, the inefficiencies that can result if most accesses to the structure use less than 100% of the data (this will result in the consumption of additional memory bandwidth).

(5) The level of temporal locality (the number of times a value is reused prior to eviction from the cache).

Additionally, the number and cost of the TLB misses must be accounted for. Fortunately, this process is somewhat simpler than it was for cache misses, since ENVELOPE assumes that TLB misses are handled one at a time by the processor and cannot be overlapped with anything. In the case where the cost of the stores is expected to be greater than the cost of the loads and prefetching is supported, the equations are as follows:

$$\text{current cache misses} = \frac{\text{PLGB} * \text{LOADS}}{\text{UCLGB} * \text{L2LSIZE}} * \frac{\text{GSIZE}}{\text{GDENSITY}}. \tag{3}$$

$$\text{current run time} = \frac{\text{PLGB} * \text{LOADS} * \text{RMTRANSACTIONS}}{\text{UCLGB} * \text{PUMRBANDWIDTH}}. \tag{4}$$

$$\text{current TLB misses} = \left(\frac{\text{PLGB} * \text{LOADS}}{\text{UCLGB} * \text{PAGESIZE}} + \frac{\text{PSGB} * \text{STORES}}{\text{UCSGB} * \text{PAGESIZE}} \right) * \frac{\text{GSIZE}}{\text{GDENSITY}}. \tag{5}$$

$$\text{TEMPBUSTRANSACTIONS} = \text{current cache misses} * \text{RMTRANSACTIONS} + \text{current TLB misses}. \tag{6}$$

$$\text{TEMPRUNTIME} = \text{current run time} * \frac{\text{GSIZE}}{\text{GDENSITY}} + \text{current TLB misses} * \text{TLATENCY} * 10^{-9}. \tag{7}$$

$$\text{current cache misses} = \frac{\text{PSGB} * \text{STORES}}{\text{UCSGB} * \text{L2LSIZE}} * \frac{\text{GSIZE}}{\text{GDENSITY}}. \tag{8}$$

$$\text{current bus transactions} = \text{current cache misses} * \text{WMTRANSACTIONS}. \tag{9}$$

$$\text{current run time} = \frac{\frac{\text{PSGB} * \text{STORES} * \text{WMTRANSACTIONS}}{\text{UCSGB}} - \text{current run time} * \text{PUMWBANDWIDTH}}{\text{UMWBANDWIDTH}}. \tag{10}$$

$$\text{TEMPRUNTIME} = \text{TEMPRUNTIME} + \text{current run time} * \frac{\text{GSIZE}}{\text{GDENSITY}}. \quad (11)$$

$$\text{total run time} = \text{total run time} + \text{TEMPRUNTIME}. \quad (12)$$

$$\text{TEMPBUSTRANSACTIONS} = \text{TEMPBUSTRANSACTIONS} + \text{current bus transactions}. \quad (13)$$

$$\text{total bus transactions} = \text{total bus transactions} + \text{TEMPBUSTRANSACTIONS}. \quad (14)$$

Other cases which are handled separately but in a similar manner are:

(1) Working set fits into cache, with loads taking longer than Stores, with prefetching supported.

(2) Working set does not fit into cache, and default back to a STRIDE-N access pattern has been specified.

(3) Working set does not fit into cache, and default back to a STRIDE-1 access pattern has been specified. In this case, all of the possible ways for computing a STRIDE-1 access pattern for a data set not fitting into cache must be handled (see subsection 3.5 for details).

(4) Working set fits into cache, but prefetching is not supported. This precludes the possibility that cache misses resulting from loads and stores can be overlapped, since cache misses resulting from loads cannot even overlap with themselves.*

3.4 STRIDE-N Memory Access Patterns

In theory, this should be the easiest of the memory access patterns to handle. In reality, some additional complications have arisen that are causing some problems. These complications will be discussed in greater detail as this section progresses. Unlike subsection 3.3, there are only two cases that need to be considered here:

(1) All of the data fits into cache. In this case, ENVELOPE assigns the cost of batching up the cache to this access pattern. The cost is calculated in a very simple manner designed to produce the smallest possible cost (e.g., the minimum number of cache misses and assuming that prefetching is supported).† Additionally, if any calculations are specified for this access pattern, their cost will also be calculated.

---

* In some cases, this might be too pessimistic, since some systems might support the overlapping of coherency traffic (e.g., write backs from cache to main memory as part of an eviction) with the handling of a cache miss. It is not clear what the exact importance of this form of overlapping would be; therefore, it is currently treated as a higher order effect and is ignored.

† In some cases, this may be overly optimistic; however, it is unlikely that any jobs other than some of the smaller benchmarks (e.g., Linpack 100 × 100) will ever fall into this case.

(2) By far, the more common case and, therefore, the one that will be considered in greater detail is the case where the amount of data is too large to fit into the cache (probably by a large amount).

In this section, the following simplifying assumptions are made when handling the second case:

(1) TLB misses do not overlap with anything.

(2) TLB misses have a fixed cost, with the page table entry coming from main memory.

(3) Effectively, no form of overlapping multiple cache misses can occur, since each of the cache misses will be associated with a TLB miss.

(4) The grouping of data into structures can reduce the number of TLB misses.*

(5) Each TLB miss is expected to have a cache miss associated with it. In reality, it is possible for an algorithm (e.g., LU decomposition) to loop through a few hundred pages of data with a STRIDE-N access pattern. In such a case, there can be significantly more TLB misses than cache misses. However, for an algorithm with a random access pattern and/or a working set involving tens of thousands (or more) of pages of data, each TLB miss should have a corresponding cache miss.†

Of all of these assumptions, the second assumption seems to be causing the most problems. If the TLB misses occur in a cyclic fashion, it is conceivable that a processor might store the page table entries in cache. On many systems, this would decrease the cost of a TLB miss by at least a factor of 2. Calculations based on benchmarking studies made for the NAS LU Benchmark (class B) indicate that on many systems, this is almost certainly happening. At the present time, the best solution to dealing with such a case is to significantly decrease the estimated cost for TLB misses when modeling such a run. It should be noted that to avoid problems with ENVELOPE's error checking and default mechanisms, the cost of a TLB miss should be at least 1 nanosecond.

---

* For codes that were analyzed by hand, this can be an important effect. However, since the author expects that most codes will be analyzed using the tool discussed in section 4, this effect is insignificant. The tool discussed in section 4 recommends input to ENVELOPE based on the output from a Perfex run. In that case, any benefit from the grouping of data into structures has already been accounted for by a corresponding reduction in the measured TLB miss rate. Such a reduction would be expressed as a decrease in the estimated percentage of the work that is mapped to the STRIDE-N access pattern.

† This is not as serious a problem as one might expect. If one assumes that most programs will be analyzed by the tool in section 4, then since that tool makes the same assumption, everything should work out. There might be some minor discrepancies due to the cache misses being handled one at a time; whereas, if they were mapped to a STRIDE-1 access pattern they could be overlapped using prefetching. However, for a well-tuned code, one can expect the number of TLB misses to be significantly smaller than the number of cache misses. In this situation, all of this becomes a higher order effect that can be safely ignored.

The resulting equations are as follows:

$$\text{current cache misses} = \frac{\text{PLGN} * \text{LOADS}}{\text{UCLGN} * \text{DSIZE} * \text{GDENSITY}}. \quad (15)$$

$$\text{current TLB misses} = \text{current cache misses}.$$

$$\text{current cache misses} = \text{current cache misses} * \left\lceil 1 + \frac{\text{GSIZE} * \text{DSIZE}}{\text{L2LSIZE}} \right\rceil. \quad (16)$$

$$\text{current bus transactions} = \text{current cache misses} * \text{RMTRANSACTIONS} + \text{current TLB misses}. \quad (17)$$

$$\text{current run time} = \begin{pmatrix} \text{current cache misses} * \text{MLATENCY} * \\ \text{RMTRANSACTIONS} + \\ \text{current TLB misses} * \text{TLATENCY} \end{pmatrix} * 10^{-9}. \quad (18)$$

$$\text{TEMPRUNTIME} = \text{current run time}. \quad (19)$$

$$\text{TEMPBUSTRANSACTIONS} = \text{current bus transactions}. \quad (20)$$

$$\text{current cache misses} = \frac{\text{PSGN} * \text{STORES}}{\text{UCSGN} * \text{DSIZE} * \text{GDENSITY}}. \quad (21)$$

$$\text{current TLB misses} = \text{current cache misses}. \quad (22)$$

$$\text{current cache misses} = \text{current cache misses} * \left\lceil 1 + \frac{\text{GSIZE} * \text{DSIZE}}{\text{L2LSIZE}} \right\rceil. \quad (23)$$

$$\text{current bus transactions} = \text{current cache misses} * \text{WMTRANSACTIONS} + \text{current TLB misses}. \quad (24)$$

$$\text{current run time} = \begin{pmatrix} \text{current cache misses} * \text{MLATENCY} * \\ \text{WMTRANSACTIONS} + \\ \text{current TLB misses} * \text{TLATENCY} \end{pmatrix} * 10^{-9}. \quad (25)$$

$$\text{TEMPRUNTIME} = \text{TEMPRUNTIME} + \text{current run time}. \quad (26)$$

$$\text{total run time} = \text{total run time} + \text{TEMPRUNTIME}. \quad (27)$$

$$\text{TEMPBUSTRANSACTIONS} = \text{TEMPBUSTRANSACTIONS} + \text{current bus transactions}. \quad (28)$$

$$\text{total bus transactions} = \text{total bus transactions} + \text{TEMPBUSTRANSACTIONS}. \quad (29)$$

$$\text{TEMPRUNTIME} = \frac{2.0 * \text{PGNMADDS} * \text{NMADDS}}{\text{UMADDS}} + \frac{\text{PGMULTIPLIES} * \text{NMULTIPLIES}}{\text{UMULTIPLIES}}. \quad (30)$$

$$\text{TEMPRUNTIME} = \text{TEMPRUNTIME} + \frac{\text{PGNADDS} * \text{NADDS}}{\text{UADDS}} + \frac{\text{PGNIOPS} * \text{NIOPS}}{\text{UIOPS}}. \tag{31}$$

$$\text{TEMPRUNTIME} = \text{MAX}\left(\text{TEMPRUNTIME}, \frac{\text{PLGN} * \text{LOADS} + \text{PSGN} * \text{STORES}}{\text{UL2BANDWIDTH}}\right). \tag{32}$$

$$\text{total run time} = \text{total run time} + \text{TEMPRUNTIME}. \tag{33}$$

3.5 STRIDE-1 Memory Access Patterns

Once again, the STRIDE-1 Memory Access pattern requires the handline of the following special cases:

(1) The data set being small enough to fit entirely in cache. Again, this case is primarily there to support some small benchmarks (e.g., Linpack 100 × 100). As seen in subsection 3.4, the cost of the cache and TLB misses have already been accounted for. Therefore, all that remains is the straightforward handling of the cost of the instructions themselves. This is done in a manner that is very similar to the last four equations in subsection 3.4.

(2) A STRIDE-1 access pattern with prefetching disabled. This implies that the cache misses do not overlap. Therefore, whether the loads or stores take longer to complete is not a concern.

(3) The cases of a STRIDE-1 access pattern with prefetching enabled. The relative costs of the loads and stores is now a concern. While these two cases must be handled separately, the resulting equations both look very similar to those used in subsection 3.3. Therefore, they will not be repeated here.

What will be looked at here is the second case. The resulting equations are as follows:

$$\text{current cache misses} = \frac{\text{PLG1} * \text{LOADS}}{\text{UCLG1} * \text{L2LSIZE}} * \frac{\text{GSIZE}}{\text{GDENSITY}}. \tag{34}$$

$$\text{current TLB misses} = \left(\frac{\text{PLG1} * \text{LOADS}}{\text{UCLG1} * \text{PAGESIZE}} + \frac{\text{GSIZE}}{\text{GDENSITY}}\right). \tag{35}$$

$$\text{current bus transactions} = \text{current cache misses} * \text{RMTRANSACTIONS} + \text{current TLB misses}. \tag{36}$$

$$\text{current run time} = \left(\begin{array}{l} \text{current cache misses} * \text{MLATENCY} * \\ \text{RMTRANSACTIONS} + \text{current TLB misses} \\ * \text{TLATENCY} \end{array} \right) * 10^{-9}. \quad (37)$$

$$\text{TEMPRUNTIME} = \text{current run time}. \quad (38)$$

$$\text{TEMPBUSTRANSACTIONS} = \text{current bus transactions}. \quad (39)$$

$$\text{current cache misses} = \frac{\text{PSG1} * \text{STORES}}{\text{UCSG1} * \text{L2LSIZE}} * \frac{\text{GSIZE}}{\text{GDENSITY}}. \quad (40)$$

$$\text{current TLB misses} = \frac{\text{PSG1} * \text{STORES}}{\text{UCSG1} * \text{PAGESIZE}} * \frac{\text{GSIZE}}{\text{GDENSITY}}. \quad (41)$$

$$\text{current bus transactions} = \text{current cache misses} * \text{WMTRANSACTIONS} + \text{current TLB misses}. \quad (42)$$

$$\text{current run time} = \left(\begin{array}{l} \text{current cache misses} * \text{MLATENCY} * \\ \text{WMTRANSACTIONS} + \\ \text{current TLB misses} * \text{TLATENCY} \end{array} \right) * 10^{-9}. \quad (43)$$

$$\text{TEMPRUNTIME} = \text{TEMPRUNTIME} + \text{current run time}. \quad (44)$$

$$\text{total run time} = \text{total run time} + \text{TEMPRUNTIME}. \quad (45)$$

$$\text{TEMPBUSTRANSACTIONS} = \text{TEMPBUSTRANSACTIONS} + \text{current bus transactions}. \quad (46)$$

$$\text{total bus transactions} = \text{total bus transactions} + \text{TEMPBUSTRANSACTIONS}. \quad (47)$$

$$\text{TEMPRUNTIME} = \frac{2.0 * \text{PG1MADDS} * \text{NMADDS}}{\text{UMADDS}} + \frac{\text{PG1MULTIPLIES} * \text{NMULTIPLIES}}{\text{UMULTIPLIES}}. \quad (48)$$

$$\text{TEMPRUNTIME} = \text{TEMPRUNTIME} + \frac{\text{PG1ADDS} * \text{NADDS}}{\text{UADDS}} + \frac{\text{PG1IOPS} * \text{NIOPS}}{\text{UIOPS}}. \quad (49)$$

$$\text{TEMPRUNTIME} = \text{MAX}\left(\text{TEMPRUNTIME}, \frac{\text{PLG1} * \text{LOADS} + \text{PSG1} * \text{STORES}}{\text{UL2BANDWIDTH}} \right). \quad (50)$$

$$\text{total run time} = \text{total run time} + \text{TEMPRUNTIME}. \quad (51)$$

4. Associated Tools

Unfortunately, most people will find it difficult, if not impossible, to analyze the usage patterns of most programs with sufficient detail for use with ENVELOPE. In an attempt to solve this problem, we have written a second program which

prompts for information from an instrumented run (e.g., Perfex on an SGI system and or the Hardware Performance Monitor on Cray vector systems). Based on a modest number of questions, it will solve a series of equations and supply a set of numbers for use with ENVELOPE. Unlike ENVELOPE, some assumptions and heuristics are used in this tool. As a result, the results may not be unique and probably will not exactly match what would be produced by a detailed analysis of the user's program. However, the results should be sufficient to allow ENVELOPE to accurately predict many aspects of the performance of the user's program (e.g., run time and performance in terms of MFLOPS). Table 4 shows a sample run of this program.

5. The Equations for the Associated Tools

This section will discuss some of the equations used by the tool which uses data from Perfex (or similar programs/libraries, e.g., **PAPI**) to simplify the job of creating an input file for ENVELOPE. This tool contains two parts. The first part is optional, and when used, will use one of two approaches (depending on the available input data) to estimate the number of floating point adds, multiplies, and multiply-add instructions that are executed during a run. It should be noted that each of these instructions actually represents a group of instructions (e.g., "adds" includes adds, subtracts, compares, as well as other less frequently used instructions such as convert, int, abs, etc.). Subsections 5.1–5.3 will discuss this part of the tool in greater detail.*

The second part of the tool calculates (or, in a few cases, provides crude estimates based on rules of thumb) most of the remaining inputs needed to describe the software to ENVELOPE. This part of the tool will be discussed in subsections 5.4–5.8.

5.1 Conventions Used in Subsections 5.2 and 5.3

The following conventions will be used in subsections 5.2 and 5.3 to simplify the equations.

* One important point to remember is that some compilers will only produce independent multiply and add instructions, other compilers will preferentially produce multiply-add instructions, and a few will produce a mix based on some optimization criteria. Furthermore, for some hardware, this will make little if any difference in the performance. However, for other systems, there might be a significant difference in performance (e.g., up to a factor of 2). Therefore, in cases where the tool estimates that a large number of independent multiply and add instructions are being used, one might want to calculate the performance based on both that set of numbers and on the assumption that the hardware is executing the instructions as though they were chained multiply-add instructions. Fortunately, in most cases, factors such as the amount of time spent on cache misses and the ratio between memory operations (loads and stores) vs. floating point operations may eliminate most of the potential difference in performance.

NADDS = number of floating point add instructions.

NFINST = number of floating point instructions.

NFLOPS = number of floating point operations.

NMADDS = number of floating point multiply-add instructions.

NMULTIPLIES = number of floating point multiply instructions.

NRECIPROCALS = number of reciprocal approximation instructions.

"$>$" = greater than.

"$\geq$" = greater than or equal to.

"$<$" = less than.

"$\leq$" = less than or equal to.

5.2 Estimating the Numbers and Types of Floating Point Instructions Using a Combination of *a priori* Data and Data From Perfex

If one has access to a count of the number of floating point operations for a run, as is frequently the case for industry standard benchmarks, then one can use that information in conjunction with the floating point instruction count from Perfex to estimate the number of times each of the three classes of instructions (adds, multiplies, multiply-adds) is executed during a run. Alternatively, by comparing the floating point instruction count from two runs (one compiled with the use of multiply-adds enabled and one compiled with their use disabled), one can also use the following equations:

If NFINST $\geq$ NFLOPS, then

$$\text{NMADDS} = 0.0,$$

$$\text{NADDS} = 0.5 * \text{NFLOPS},$$

and

$$\text{NMULTIPLIES} = \text{NADDS}. \tag{52}$$

Otherwise, if FLINST $\leq 0.5 *$ NFLOPS, then

$$\text{NMADDS} = 0.5 * \text{NFLOPS},$$

$$\text{NADDS} = 0.0,$$

and

$$\text{NMULTIPLIES} = 0.0. \tag{53}$$

Otherwise,

$$\text{NMADDS} = \text{NFLOPS} - \text{NFINST},$$

$$\text{NADDS} = \text{NFINST} - \text{NMADDS},$$

and

$$NMULTIPLIES = 0.0.* \quad (54)$$

It is important to note that the equations being used have been made more robust by eliminating the assumption that the compiler produced a floating point operation count that was identical to that produced by a priori knowledge. In some cases, an optimizing compiler might do slightly better. In other cases, an optimizing compiler might even add floating point operations if it thought that the efficient use of multiply-add instructions would improve the overall performance of the code. By not relying on this assumption, one can be certain that none of the operation counts will ever be negative or exceed the specified number of floating point operations.

5.3 Estimating the Numbers and Types of Floating Point Instructions Using a Combination of Data From HPM and Perfex

If one has access to both the output from the hardware performance monitor on a Cray Research vector processor (e.g., C90) and the floating point instruction count from Perfex, one can estimate the number of times each of the three classes of instructions (adds, multiplies, multiply-adds) is executed during a run. In this case, the reciprocal approximation instruction from the Cray vector processor will be lumped in with the multiply instructions. The rationale for this is to treat the combination of the reciprocal approximation with the additional refinement step as a divide instruction. ENVELOPE suggests that divides be included with the multiply instructions with an appropriate weighting factor. Effectively, this is what we are doing in the following equations:

If $NADDS + NMULTIPLIES + NRECIPROCALS \leq NFINST$, then

$$NMULTIPLIES = NMULTIPLIES + NRECIPROCALS,$$

$$NMADDS = 0.0, \quad (55)$$

and

$$NADDS \text{ remains unchanged.}$$

Otherwise, if $NADDS + NMULTIPLIES + NRECIPROCALS \geq 0.5 * NFINST$, then

$$NMADDS = \text{minimum of (NADDS or NMULTIPLIES)},$$

$$NADDS = NADDS - NMADDS,$$

---

* For most of today's processors, the cost of a floating point add is the same as the cost of a floating point multiply. Therefore, assigning all of the excess floating point instructions to the floating point adds will not effect the results produced by ENVELOPE. However, on some older processors such as the MIPS R4000/R4400 and the KSR1, this assumption is no longer valid. On these machines, there may be a difference in the estimated performance, and one might want to determine what the bounds on this performance are by running ENVELOPE twice—once with all of the excess instructions classified as floating point adds and the second time with all of the excess instructions classified as floating point multiplies.

and

NMULTIPLIES = NMULTIPLIES + NRECIPROCALS - NMADDS. (56)

Otherwise, if NADDS < MULTIPLIES, then

NMADDS = minimum of (NRECIPROCALS + NADDS + NMULTIPLIES - NFINST or NADDS),

NADDS = NADDS - NMADDS,

and

NMULTIPLIES = NMULTIPLIES + NRECIPROCALS - NMADDS. (57)

Otherwise,

NMADDS = minimum of (NRECIPROCALS + NADDS + NMULTIPLIES - NFINST or NMULTIPLIES),

NADDS = NADDS - NMADDS,

and

NMULTIPLIES = NMULTIPLIES + NRECIPROCALS - NMADDS. (58)

Again, one can see that we were careful to handle the situations where the numbers do not add up. However, to the extent that numbers do add up, we make the assumption that programs take advantage of chained multiplies and adds on the Cray vector processors. Therefore, these chained operations should be translated into multiply-add instructions for the purposes of running ENVELOPE.

5.4 Conventions and Approximations Used in Subsections 5.5–5.8

In subsections 5.5–5.8, the following approximations are made:

(1) Since Perfex is being used to return data on a complete run, the data is not broken down by subroutine, let alone memory access pattern. Therefore, for each type of memory access pattern (e.g., STRIDE-1), this tool assumes that the same percentage of loads as stores are used in a section. This does not mean that the STRIDE-1 access pattern has both a billion loads and a billion stores. Rather, it means that if 5% of the total loads exhibit the STRIDE-1 access pattern, we will assume that 5% of the total stores will also exhibit this access pattern.

(2) For the same reasons as in (1), we will assume that the same percentage of multiply-adds, adds, multiplies, and integer operations unrelated to address calculations (normally, the number of these calculations will be zeroed out for floating point intensive applications) are used for each type of memory access pattern. Furthermore, we will assume that this percentage is the same as that calculated in (1).

(3) In accordance with the way ENVELOPE is set up, we will assume that each TLB miss has a cache miss associated with it. This need not be the case if the cache line is still in the cache, as can happen with a STRIDE-N access pattern executed with a cyclic basis. However, if the number of data items is too large or if the access pattern is actually fairly random, then this assumption is correct. In either case, since Perfex does not provide a way to distinguish between the two cases, both ENVELOPE and this tool have been set up to function in this manner. Therefore, this should not result in any problems.

(4) The recommended values for the group size and group density are 1 unless known otherwise, in which case, one might want to use the value 1 since the consequences of the grouping of data were already factored into the output of Perfex and would be difficult to compensate for at this point.

(5) The number of integer loads and stores is negligible for a floating point intensive code. So the time required for them can be ignored.

(6) Data reuse at the register level has already been factored in by the compiler, having eliminated the loads and stores at compile time. Therefore, there were no "theoretical" loads and stores to be accounted for by the *SVDEDICATED and *SVGENERAL input parameters. The recommended output values will then be 1.0 for the "Used" values indicating no data reuse and 0.0 for the "percentage" values indicating no work is attributed to this access pattern.

In subsections 5.5–5.8, the following conventions will be used:

ADJMEM = the adjusted number of memory operations.
C1L1 = the number of L1 cache misses attributed to anything other than a STRIDE-N (or random) access pattern. This can result from either a STRIDE-1 access pattern or a second larger working set that fits into the L2 but not the L1 cache.
C1L2 = the number of L2 cache misses attributed to anything other than a STRIDE-N (or random) access pattern. Primarily, this is expected to be the result of a STRIDE-1 access pattern.
CGBL1 = The L1 cache misses associated with a larger working set (e.g., from a blocked access pattern involving large "global" arrays).
CN = the number of cache misses attributed to a STRIDE-N (or random) access pattern.
DSIZE = the size of the data item in bytes (usually 8).
DPERL1 = L1LSIZE/DSIZE = the number of data elements per L1 cache line.
DPERL2 = L2LSIZE/DSIZE = the number of data elements per L2 cache line
L1LSIZE = the size of the cache line in the L1 cache.
L1MISS = the number of L1 cache misses.
L1PERL2 = L2LSIZE/L1LSIZE = the number of L1 cache lines per L2 cache lines.
L2LSIZE = the size of the cache line in the L2 cache.

L2MISS = the number of L2 cache misses.
L2PERPAGE = PAGESIZE/L2LSIZE = the number of L2 cache lines per page.
NLOADS = the number of LOADS that graduated (completed).
NSTORES = the number of STORES that graduated.
PAGESIZE = the page size in bytes.
PERG1 = The percentage of the memory operations with a STRIDE-1 access pattern. Initially, this value will be set to 0.0.
PERGB = The percentage of the memory operations associated with the blocked memory access pattern associated with the larger working set.
PERGN = The percentage of the memory operations with a STRIDE-N access pattern.
PERSA = The percentage of the memory operations associated with the use of small scratch arrays in the smaller working set.
TLBMISS = the number of TLB misses.
USEG1 = The data use/reuse for the STRIDE-1 access pattern. Initially, this value will be set to 1.0, indicating no reuse. However, if the value for PERG1 changes from its initial value of 0.0, this number will be recalculated.
USEGB = The data use/reuse associated with the blocked memory access pattern associated with the larger working set.
USEGN = The data use/reuse for the STRIDE-N access pattern. This will always be hardwired as 1.0, indicating no reuse. This does not really matter since any reuse that does occur will simply be charged to another access pattern in a manner that does not result in additional TLB or cache misses.

5.5 Solving for the STRIDE-N Access Pattern Parameters

The next stage of the process is to solve for the STRIDE-N access parameters, since that will tell us how many L2 misses remain to be allocated among the remaining access patterns. Again, reasonable checks will be made and, if necessary, the values will be adjusted accordingly. The need for this can have any number of sources (e.g., extraneous cache and TLB misses due to timesharing a processor or alternatively process migration on a shared memory SMP). The tool will now solve the following system of equations:

$$\text{L2MISS} = \text{C1L2} + \text{CN}. \tag{59}$$

$$\text{TLBMISS} = \frac{\text{C1L2}}{\text{L2PERPAGE}} + \text{CN}. \tag{60}$$

Solving for C1L2 and CN, one comes up with the following equations:

$$\text{C1L2} = \frac{\text{L2MISS} - \text{TLBMISS}}{1.0 - \frac{1.0}{\text{L2PERPAGE}}}. \tag{61}$$

$$\text{CN} = \text{L2MISS} - \text{C1L2}. \tag{62}$$

Performing the sanity checks, one ends up with

If C1L2 < 0.0, then

$$C1L2 = 0.0 \text{ and}$$

$$CN = L2MISS. \tag{63}$$

Otherwise, if CN < 0.0, then

$$C1L2 = L2MISS \text{ and}$$

$$CN = 0.0. \tag{64}$$

$$ADJMEM = NLOADS + NSTORES. \tag{65}$$

$$PERGN = \frac{CN}{ADJMEM}. \tag{66}$$

$$ADJMEM = ADJMEM - CN. \tag{67}$$

$$C1L1 = L1MISS - CN. \tag{68}$$

$$CGBL1 = C1L1 - C1L2 * L1PERL2. \tag{69}$$

5.6 Solving for the Blocked Access Pattern Parameters

Now that the STRIDE-N access pattern has been accounted for and the number of memory operations and cache misses that remain to be accounted for is known, we proceed to the question of the existence of a large working set which will live out of the L2 cache. The heuristic that will be used at this point is somewhat arbitrary but is based on the concept that unless there is a reasonable amount of data reuse, one cannot say that a working set exists. If $\frac{C1L1}{L1PERL2 * C1L2} \geq 4.0$, then we have a large working set, and the following equations are used:

$$PERG1 = 0.0. \tag{70}$$

$$USEG1 = 1.0. \tag{71}$$

This implies that all of the remaining L2 cache misses will be charged to the blocked access pattern, with no work assigned to a STRIDE-1 access pattern. This is a somewhat arbitrary assignment. However, since both access patterns will produce the same ratio between TLB misses and L2 cache misses, this should not be a problem as long as the working set fits into the cache. If one moves onto a system that lacks an L2 cache or where the cache is too small, then one needs to specify if ENVELOPE is to treat the resulting access pattern as if it is STRIDE-1 or STRIDE-N. The recommended default when using this tool is STRIDE-1, which is specified by negating the estimated size of the working set to be discussed in more detail in section 5.8.

The tool now assumes that all of the L1 cache misses are the result of this larger working set, since if a smaller working set also exists, it will live out of the L1 cache. As such, the smaller working set is expected to have a negligible cache miss rate. This corresponds to the use of small scratch arrays (the SA input parameters for ENVELOPE).

$$\text{ADJMEM} = \text{ADJMEM} - \text{C1L1} * \text{DPERL1}. \tag{72}$$

If ADJMEM < 0.0, then

$$\text{ADJMEM} = 0.0. \tag{73}$$

$$\text{PERGB} = \frac{\text{C1L1} * \text{DPERL1}}{\text{NLOADS} + \text{NSTORES}}. \tag{74}$$

$$\text{USEGB} = \frac{\text{C1L1}}{\text{L1PERL2} * \text{C1L2}}. \tag{75}$$

$$\text{PERSA} = \frac{\text{ADJMEM}}{\text{NLOADS} + \text{NSTORES}}. \tag{76}$$

The recommended size for the large working set is 1.0 **MB**, with a STRIDE-1 access pattern if the cache is too small. The choice of 1.0 MB is somewhat arbitrary but is based on most SGI systems in recent years using a L2 cache size of 1–8 MB. Furthermore, most of the competing systems, when equipped with a large cache, also have a cache size of at least 1 MB. However, experience has indicated that some of the NAS benchmarks have a large working set small enough to fit in the caches of the Cray T3E and the IBM SP with Power 2 Super Chips. Therefore, prudence dictates that one might want to compare the predicted performance to the measured performance on one of these systems in an attempt to fine-tune this parameter. All we know for certain is that the size of the larger working set is somewhere between the size of the L1 and L2 caches. This concludes the handling of the situation in which a large working set occurs. Subsections 5.7 and 5.8 only apply to the situation where a working set is either missing or not very effective.

5.7 Checking for the Case of a Small Working Set Without a Large Working Set

The tool starts out by setting the parameters that describe the larger working set, causing that access pattern to be skipped by ENVELOPE.

$$\text{PERGB} = 0.0. \tag{77}$$

$$\text{USEGB} = 1.0. \tag{78}$$

Once again, the tool uses a heuristic to check to see if a small working set exists and is effective.

If $\frac{\text{ADJMEM}}{\text{DPERL1} * \text{CGBL1}} \geq 4.0$, then we have a small working set. This means that there is a cache resident small scratch array. The tool now calculates what percentage of the memory operations involve this small working set and what percentage needs to be mapped to the STRIDE-1 access pattern to achieve the correct number of L2 cache misses and TLB misses. It should be noted that the small working set is assumed to have an insignificant number of L2 cache misses and TLB misses associated with it. Since the STRIDE-1 access pattern is being used only to the extent necessary to account for the L2 cache misses and the TLB misses, it will be assigned a data use/reuse value of 1.0, indicating that all data reuse is associated with the small working set.

$$\text{PERSA} = \frac{\text{ADJMEM} - \text{C1L2} * \text{DPERL2}}{\text{NLOADS} + \text{NSTORES}}. \tag{79}$$

If PERSA > 1.0, then

$$\text{PERSA} = 1.0. \tag{80}$$

$$\text{PERG1} = 1.0 - \text{PERSA} - \text{PERGN}. \tag{81}$$

If PERG1 > 1.0, then

$$\text{PERG1} = 1.0. \tag{82}$$

If PERSA < 0.0, then

$$\text{PERSA} = 0.0. \tag{83}$$

$$\text{USEG1} = 1.0. \tag{84}$$

The last remaining value is the estimated size of the smaller working set. All we know for certain is that it has to fit into the 32 **kB** cache of the MIPS R10K or R12K processor of the system that has been used. It probably is somewhat smaller than that, so the tool recommends the value of 12 kB, which is a safe number for almost all of the RISC processors made since 1990. The tool has now completed its task, and subsection 5.8 should be skipped.

5.8 Handling the Case Where No Working Sets Exist

The tool has now determined that no working sets exist, so all of the data access must be mapped to either a STRIDE-N access pattern or a STRIDE-1 access pattern. In subsection 5.5, the portion mapped to the STRIDE-N access pattern was calculated, leaving the STRIDE-1 access pattern to be handled now.

$$\text{PERSA} = 0.0. \tag{85}$$

$$\text{PERGG1} = 1.0 - \text{PERGN}. \tag{86}$$

$$\text{USEG1} = \frac{\text{ADJMEM}}{\text{DPERL2} * \text{C1L2}}. \tag{87}$$

The only complicated part of this is that any data reuse that occurs must now be mapped to the STRIDE-1 access pattern, as was done in the last equation. This concludes the discussion of the equations and logic behind this tool.

6. Future Work

Work is currently under way to improve the usability of this code. Additionally, research has been initiated to try and identify what characteristics of a parallel code need to be taken into consideration when estimating the performance of a parallel program. Unfortunately, our initial experience in this area indicates that this is a highly complex problem that is probably too difficult to tackle in the general case. We hope that in the future, we will be able to produce useful simulators for some of the more commonly found cases.

7. Results and Conclusions

We have created an entirely new simulator based on Back-of-the-ENVELOPE calculations that is capable of simulating the performance of computationally intensive workloads in a short fixed amount of time. An associated tool that makes the simulator friendlier to use has also been discussed. Experience with using ENVELOPE has shown that in almost all cases, it can accurately predict the performance of the user's code to within a factor of 2 of the measured value. Furthermore, in many cases, we were able to achieve agreement with experimental results to within ±10–15%.

Table 1. Input parameters for an IBM SP with 375 MHz for the Linpack Power 3 Thin SMP nodes 100 × 100 benchmark.

Input Data	Annotations Produced by ENVELOPE for ENVELOPE.INPUTS	Additional Comments
400.00	memory latency in NS	Characteristics of the hardware.
8.000000	cache size in MB	
128	cache line size in bytes	
8000.000	cache bandwidth for hits in MB/second	
4	page size in kB	
800.00	TLB latency in NS	
1000.0000	memory bandwidth for reads in MB/second	
1000.0000	memory bandwidth for writes in MB/second	
1000.0000	memory bandwidth for a mix of reads/writes MB/second	
2	the number of "bus" transactions per write miss	
1500.0000	the peak speed when performing MADDS	
750.0000	the peak speed when performing Multiplies	
750.0000	the peak speed when performing Adds	
m	the pipeline depth	
y	Out-of-Order execution	
y	Register Renaming	
10	the maximum number of outstanding prefetches/cache misses	
1500.0000	peak rate for integer operations	
8	size of the data item in bytes	General characteristics of the software.
7.6700002E-03	the amount of data being loaded into the processor in GB	
2.5599999E-03	the amount of data being stored by the processor in GB	
343000.0	the number of madds	
0.0000000E+00	the number of multiplies, etc.	
0.0000000E+00	the number of adds, etc.	
7.9999998E-02	the memory footprint (RSS)	
1.000000	is the size of the group	
1.000000	is the density of the group	
0.0000000E+00	the number of integer operations	

Table 1. Input parameters for an IBM SP with 375 MHz for the Linpack Power 3 Thin SMP nodes 100 × 100 benchmark (continued).

Input Data	Annotations Produced by ENVELOPE for ENVELOPE.INPUTS	Additional Comments
1.000000	Used RLSVDEDICATED	Operations only involving data held in registers.
0.0000000E+00	Percentage LSVDEDICATED	
1.000000	Used RSSVDEDICATED	
0.0000000E+00	Percentage SSVDEDICATED	
0.0000000E+00	Percentage SVMADDS	
0.0000000E+00	Percentage SVMULTIPLIES	
0.0000000E+00	Percentage SVADDS	
0.0000000E+00	Percentage SVIOPS	
100.0000	Used RLSVGENERAL	Other operations involving data held in registers.
33.33333	Percentage LSVGENERAL	
1.000000	Used RSSVGENERAL	
0.0000000E+00	Percentage SSVGENERAL	
1.000000	Used RLSADEDICATED	Operations only involving scratch arrays and data held in registers.
0.0000000E+00	Percentage LSADEDICATED	
1.000000	Used RSSADEDICATED	
0.0000000E+00	Percentage SSADEDICATED	
0.0000000E+00	Percentage SAMADDS	
0.0000000E+00	Percentage SAMULTIPLIES	
0.0000000E+00	Percentage SAADDS	
0.0000000E+00	Percentage SAIOPS	
1.000000	Used RLSAGENERAL	Other operations involving scratch arrays.
0.0000000E+00	Percentage LSAGENERAL	
1.000000	Used RSSAGENERAL	
0.0000000E+00	Percentage SSAGENERAL	
1.170000E-02	the scratch array memory footprint	Working set size.

Table 1. Input parameters for an IBM SP with 375 MHz for the Linpack Power 3 Thin SMP nodes 100 × 100 benchmark (continued).

Input Data	Annotations Produced by ENVELOPE for ENVELOPE.INPUTS	Additional Comments
1.000000	Used CLGB	Operations involving either a blocked access pattern or a second, larger working set.
0.0000000E+00	Percentage LGB	
1.000000	Used CSGB	
0.0000000E+00	Percentage SGB	
0.0000000E+00	Percentage GBMADDS	
0.0000000E+00	Percentage GBMULTIPLIES	
0.0000000E+00	Percentage GBADDS	
0.0000000E+00	Percentage GBIOPS	
1.000000	the block size	
1.000000	Used CLGN	Operations involving a Stride N access pattern.
0.0000000E+00	Percentage LGN	
1.000000	Used CSGN	
0.0000000E+00	Percentage SGN	
0.0000000E+00	Percentage GNMADDS	
0.0000000E+00	Percentage GNMULTIPLIES	
0.0000000E+00	Percentage GNADDS	
0.0000000E+00	Percentage GNIOPS	
1.670000	Used CLG1	Operations involving a Stride 1 access pattern.
67.00000	Percentage LG1	
1.000000	Used CSG1	
100.0000	Percentage SG1	
100.0000	Percentage G1MADDS	
100.0000	Percentage G1MULTIPLIES	
100.0000	Percentage G1ADDS	
100.0000	Percentage G1IOPS	

Table 2. Input parameters for an IBM SP with 375-MHz Power 3 Thin SMP nodes for the CG NAS benchmark (class B using MPI) with prefetching "disabled."

Input Data	Annotations Produced by ENVELOPE for ENVELOPE.INPUTS	Additional Comments
400.0000	memory latency in NS	Characteristics of the hardware.
8.000000	cache size in MB	
128	cache line size in bytes	
8000.000	cache bandwidth for hits in MB/second	
4	page size in kB	
800.0000	TLB latency in NS	
1000.000	memory bandwidth for reads in MB/second	
1000.000	memory bandwidth for writes in MB/second	
1000.000	memory bandwidth for a mix of reads/writes MB/second	
2	the number of "bus" transactions per write miss	
1500.000	the peak speed when performing MADDS	
750.0000	the peak speed when performing Multiplies	
750.0000	the peak speed when performing Adds	
m	the pipeline depth	
y	Out-of-Order execution	
y	Register Renaming	
1	the maximum number of outstanding prefetches/cache misses	
1500.000	peak rate for integer operations	
8	size of the data item in bytes	General characteristics of the software.
628.0000	the amount of data being loaded into the processor in GB	
8.407000	the amount of data being stored by the processor in GB	
2.8706488E+10	the number of madds	
0.0000000E+00	the number of multiplies, etc.	
0.0000000E+00	the number of adds, etc.	
776.0000	the memory footprint (RSS)	
1.000000	is the size of the group	
1.000000	is the density of the group	
0.0000000E+00	the number of integer operations	

Table 2. Input parameters for an IBM SP with 375-MHz Power 3 Thin SMP nodes for the CG NAS benchmark (class B using MPI) with prefetching "disabled" (continued).

Input Data	Annotations Produced by ENVELOPE for ENVELOPE.INPUTS	Additional Comments
1.000000	Used RLSVDEDICATED	Operations only involving data held in registers.
0.0000000E+00	Percentage LSVDEDICATED	
1.000000	Used RSSVDEDICATED	
0.0000000E+00	Percentage SSVDEDICATED	
0.0000000E+00	Percentage SVMADDS	
0.0000000E+00	Percentage SVMULTIPLIES	
0.0000000E+00	Percentage SVADDS	
0.0000000E+00	Percentage SVIOPS	
1.000000	Used RLSVGENERAL	Other operations involving data held in registers.
0.0000000E+00	Percentage LSVGENERAL	
1.000000	Used RSSVGENERAL	
0.0000000E+00	Percentage SSVGENERAL	
1.000000	Used RLSADEDICATED	Operations only involving scratch arrays and data held in registers.
0.0000000E+00	Percentage LSADEDICATED	
1.000000	Used RSSADEDICATED	
0.0000000E+00	Percentage SSADEDICATED	
0.0000000E+00	Percentage SAMADDS	
0.0000000E+00	Percentage SAMULTIPLIES	
0.0000000E+00	Percentage SAADDS	
0.0000000E+00	Percentage SAIOPS	
1.000000	Used RLSAGENERAL	Other operations involving scratch arrays.
0.0000000E+00	Percentage LSAGENERAL	
1.000000	Used RSSAGENERAL	
0.0000000E+00	Percentage SSAGENERAL	
1.170000E-02	the scratch array memory footprint	Working set size.

Table 2. Input parameters for an IBM SP with 375-MHz Power 3 Thin SMP nodes for the CG NAS benchmark (class B using MPI) with prefetching "disabled" (continued).

Input Data	Annotations Produced by ENVELOPE for ENVELOPE.INPUTS	Additional Comments
1.000000	Used CLGB	Operations involving either a blocked access pattern or a second, larger working set.
0.0000000E+00	Percentage LGB	
1.000000	Used CSGB	
0.0000000E+00	Percentage SGB	
0.0000000E+00	Percentage GBMADDS	
0.0000000E+00	Percentage GBMULTIPLIES	
0.0000000E+00	Percentage GBADDS	
0.0000000E+00	Percentage GBIOPS	
-0.1100000	the block size	
1.000000	Used CLGN	Operations involving a Stride N access pattern.
2.2634468E-03	Percentage LGN	
1.000000	Used CSGN	
2.2634468E-03	Percentage SGN	
2.2634468E-03	Percentage GNMADDS	
2.2634468E-03	Percentage GNMULTIPLIES	
2.2634468E-03	Percentage GNADDS	
2.2634468E-03	Percentage GNIOPS	
2.055018	Used CLG1	Operations involving a Stride 1 access pattern.
99.99773	Percentage LG1	
2.055018	Used CSG1	
99.99773	Percentage SG1	
99.99773	Percentage G1MADDS	
99.99773	Percentage G1MULTIPLIES	
99.99773	Percentage G1ADDS	
99.99773	Percentage G1IOPS	

Table 3. A comparison of predicted results from ENVELOPE to measured results.

System	Benchmark	Predicted Speed (MFLOPS)	Measured Speed (MFLOPS)	Measured / Predicted	Source
02K-195	Linpack 100 × 100	132	114	0.86	[3]
02K-195	BT	64	55	0.86	[5]
02K-195	CG	43	39	0.91	[5]
02K-195	LU	74	92	1.24	[5]
02K-195	LU	108	92	0.85	[5], assuming a dedicated node and TLB misses to L2 cache.
02K-195	LU	74	44	0.59	[6]
02K-195	SP	50	42	0.84	[5]
02K-195	SP	63	42	0.67	[5], assuming a dedicated node.
02K-195	F3D-shared	205	177	0.86	
02K-195	F3D-distributed	55	50	0.91	[7]
02K-195	CTH	152	87	0.57	[8]
02K-195	CTH	152	87	0.57	[8], assuming that prefetching is ineffective.
02K-250	BT	68	79	1.16	[6]
02K-250	CG	47	38	0.81	[9]
02K-250	LU	78	85	1.09	[9]
02K-250	LU	117	85	0.73	[9], assuming a dedicated node and TLB misses to L2 cache.
02K-250	SP	52	68	1.31	[9]
02K-250	SP	67	68	1.01	[9], assuming a dedicated node.

Table 3. A comparison of predicted results from ENVELOPE to measured results (continued).

System	Benchmark	Predicted Speed (MFLOPS)	Measured Speed (MFLOPS)	Measured / Predicted	Source
02K-300	Linpack 100 × 100	179	173	0.97	[3]
02K-300	BT	70	72	1.03	[6]
02K-300	CG	49	44	0.90	[6]
02K-300	LU	81	88	1.09	[6]
02K-300	LU	124	88	0.71	[6], assuming a dedicated node and TLB misses to L2 cache.
02K-300	SP	54	69	1.28	[6]
02K-300	SP	70	69	0.99	[6], assuming a dedicated node.
02K-300	F3D-shared	264	248	0.94	
02K-300	F3D-distributed	59	62	1.05	[7]
02K-300	CTH	208	125	0.60	[8]
02K-300	CTH	209	125	0.60	[8], assuming that prefetching is ineffective.
03K-400	Linpack 100 × 100	273	199	0.73	[10]
03K-400	BT	128	130	1.02	[10]
03K-400	CG	90	69	0.77	[10]
03K-400	CG	53	69	1.30	[10], assuming that prefetching is ineffective.
03K-400	LU	127	224	1.76	[10]
03K-400	LU	179	224	1.25	[10], assuming a dedicated node and TLB misses to L2 cache.
03K-400	SP	74	122	1.65	[10]
03K-400	SP	127	122	0.96	[10], assuming a dedicated node.
03K-400	F3D-shared	397	377	0.95	
03K-400	CTH	292	193	0.66	[8]
03K-400	CTH	252	193	0.77	[8], assuming that prefetching is ineffective.
SUN E10000	F3D-shared	225	180	0.80	
SUN E10000	CTH	187	97	0.52	[8]
SUN E10000	CTH	156	97	0.62	[8], assuming that prefetching is ineffective.

Table 3. A comparison of predicted results from ENVELOPE to measured results (continued).

System	Benchmark	Predicted Speed (MFLOPS)	Measured Speed (MFLOPS)	Measured / Predicted	Source
T3E-1200	BT	156	67	0.43	[5]
T3E-1200	BT	54	67	1.24	[5], assuming poor use of prefetching.
T3E-1200	CG	92	10	0.11	[5]
T3E-1200	CG	32	10	0.31	[5], assuming poor use of prefetching.
T3E-1200	LU	152	79	0.52	[5]
T3E-1200	SP	91	50	0.55	[5]
T3E-1200	SP	49	50	1.02	[5], assuming poor use of prefetching.
T3E-1200	F3D-distributed	127	57	0.45	[7]
T3E-1200	F3D-distributed	47	57	1.21	[7], assuming poor use of prefetching.
T3E-900	BT	138	58	0.42	[5]
T3E-900	BT	52	58	1.12	[5], assuming poor use of prefetching.
T3E-900	CG	81	11	0.14	[5]
T3E-900	CG	30	11	0.37	[5], assuming poor use of prefetching.
T3E-900	LU	137	66	0.48	[5]
T3E-900	SP	84	44	0.52	[5]
T3E-900	SP	47	44	0.94	[5], assuming poor use of prefetching.
T3E-900	F3D-distributed	118	43	0.36	[7]
P2-66.7	CTH	86	63	0.73	[8]
P2-66.7	CTH	77	63	0.82	[8], assuming that prefetching is ineffective.
P2SC-120	Linpack 100×100	198	233	1.18	[3]
P2SC-120	Linpack 100×100	198	233	1.18	[3], TWEAKED input that the pipeline depth was short.
P2SC-120	BT	148	104	0.70	[5]
P2SC-120	LU	126	97	0.77	[5]
P2SC-120	LU	144	97	0.67	[5], assuming TLB misses to cache.
P2SC-120	SP	68	72	1.06	[5]
P2SC-135	Linpack 100×100	220	265	1.20	[3]

Table 3. A comparison of predicted results from ENVELOPE to measured results (continued).

System	Benchmark	Predicted Speed (MFLOPS)	Measured Speed (MFLOPS)	Measured / Predicted	Source
P2SC-160	Linpack 100 × 100	256	315	1.23	[3]
P2SC-160	BT	169	131	0.78	[5]
P2SC-160	CG	112	31	0.28	[5]
P2SC-160	CG	80	31	0.39	[5], assuming poor use of prefetching.
P2SC-160	LU	138	129	0.93	[4]
P2SC-160	LU	160	129	0.81	[4], assuming TLB misses to cache.
P2SC-160	SP	73	92	1.26	[5]
P2SC-160	F3D-distributed	123	33	0.27	[7]
P2SC-160	F3D-distributed	92	33	0.36	[7], assuming poor use of prefetching.
POWER3-200	Linpack 100 × 100	295	233	0.79	[11]
POWER3-200	BT	147	108	0.73	[4]
POWER3-200	BT	135	108	0.80	[4], assuming poor use of prefetching.
POWER3-200	CG	86	44	0.51	[4]
POWER3-200	CG	79	44	0.56	[4], assuming poor use of prefetching.
POWER3-200	LU	152	147	0.97	[4]
POWER3-200	LU	222	147	0.66	[4], assuming a dedicated node and TLB misses to L2 cache.
POWER3-200	SP	94	84	0.89	[4]
POWER3-222	Linpack 100 × 100	363	250	0.69	[3]
POWER3-222	BT	215	105	0.49	[12] (the measured results are for the class A data set).
POWER3-222	BT	71	105	1.48	[12], assuming poor use of prefetching.
POWER3-222	CG	136	74	0.54	[13]
POWER3-222	CG	42	74	1.76	[13], assuming poor use of prefetching.
POWER3-222	LU	139	141	1.01	[12] (the measured results are for the class A data set).
POWER3-222	LU	174	141	0.81	[12], assuming a dedicated node and TLB misses to L2 cache.
POWER3-222	SP	66	77	1.17	[12] (the measured results are for the class A data set).

Table 3. A comparison of predicted results from ENVELOPE to measured results (continued).

System	Benchmark	Predicted Speed (MFLOPS)	Measured Speed (MFLOPS)	Measured / Predicted	Source
POWER3-375-Thin	Linpack 100 × 100	357	426	1.19	[14]
POWER3-375-Thin	BT	90	74	0.82	[4]
POWER3-375-Thin	BT	90	74	0.82	[4], assuming poor use of prefetching.
POWER3-375-Thin	CG	54	45	0.83	[4]
POWER3-375-Thin	CG	54	45	0.83	[4], assuming poor use of prefetching.
POWER3-375-Thin	LU	92	224	2.43	[4]
POWER3-375-Thin	LU	238	224	0.95	[4], assuming a dedicated node and TLB misses to L2 cache.
POWER3-375-Thin	SP	56	86	1.54	[4]
POWER3-375-Thin	SP	79	86	1.09	[4], assuming a dedicated node.
POWER3-375-High	Linpack 100 × 100	505	424	0.84	[14]
POWER3-375-High	CG	110	56	0.51	[15]
POWER3-375-High	CG	45	45	1.24	[15], assuming poor use of prefetching.
POWER3-375-High	LU	129	288	2.23	[15]
POWER3-375-High	LU	195	288	1.53	[15], assuming a dedicated node and TLB misses to L2 cache.
POWER3-375-High	CTH	409	272	0.67	[8]
POWER3-375-High	CTH	309	272	0.88	[8], assuming that prefetching is ineffective.
DS20-500	Linpack 100 × 100	336	440/270	1.31/0.82	[3] (two significantly different values were reported).
DS20-500	Linpack 100 × 100	358	440/270	1.23/0.75	[3], assuming a dedicated system.
DS20-500	BT	204	206/174	1.01/0.85	[16] (TRU64/Linux) class A data set.
DS20-500	CG	122	95/90	0.78/0.74	[16] (TRU64/Linux) class A data set.
DS20-500	CG	47	95/90	2.02/1.91	[16], assuming poor use of prefetching.
DS20-500	LU	187	146/137	0.78/0.73	[16] (TRU64/Linux) class A data set.
DS20-500	LU	211	146/137	0.66/0.62	[16], assuming TLB misses to L2 cache.
DS20-500	SP	107	192/127	1.79/1.19	[16] (TRU64/Linux) class A data set.

Table 3. A comparison of predicted results from ENVELOPE to measured results (continued).

System	Benchmark	Predicted Speed (MFLOPS)	Measured Speed (MFLOPS)	Measured / Predicted	Source
ES40-500	BT	204	194/175	0.95/0.86	[16] (TRU64/Linux) class A data set.
ES40-500	CG	122	90/87	0.74/0.72	[16] (TRU64/Linux) class A data set.
ES40-500	CG	47	90/87	1.91/1.85	[16], assuming poor use of prefetching.
ES40-500	LU	187	190/132	1.02/0.71	[16] (TRU64/Linux) class A data set.
ES40-500	LU	211	190/132	0.90/0.63	[16], assuming TLB misses to L2 cache.
ES40-500	SP	135	161/142	1.19/1.05	[16] (TRU64/Linux) class A data set.
ES40-667	L100	430	561	1.30	[3]
ES40-667	L100	466	561	1.20	[3], assuming a dedicated system.
ES40-667	BT	226	150	0.66	[17]
ES40-667	CG	137	110	0.80	[18]
ES40-667	CG	49	110	2.24	[18], assuming poor use of prefetching.
ES40-667	LU	202	250	1.24	[17]
ES40-667	LU	231	250	1.08	[17], assuming TLB misses to L2 cache.
ES40-667	SP	113	150	1.33	[17]
ES40-667	SP	127	150	1.18	[17], assuming a dedicated system.
Pentium-II-450	Linpack 100×100	116	98	0.84	[3]
Pentium-II-450	BT	39	56	1.44	[19]
Pentium-II-450	CG	27	28	1.04	[19]
Pentium-II-450	CG	27	28	1.04	[19], assuming poor use of prefetching.
Pentium-II-450	LU	50	100	2.00	[19]
Pentium-II-450	LU	102	100	0.98	[19], assuming a dedicated node and TLB misses to L2 cache.
Pentium-II-450	SP	47	47	47	[19]
Pentium-III-733	BT	41	58	1.41	[19]
Pentium-III-733	CG	29	38	1.31	[19]
Pentium-III-733	CG	29	38	1.31	[19], assuming poor use of prefetching.
Pentium-III-733	LU	53	106	2.00	[19]
Pentium-III-733	LU	116	106	0.91	[19], assuming a dedicated node and TLB misses to L2 cache.
Pentium-III-733	SP	38	39	1.03	[19]

Table 3. A comparison of predicted results from ENVELOPE to measured results (continued).

System	Benchmark	Predicted Speed (MFLOPS)	Measured Speed (MFLOPS)	Measured / Predicted	Source
HP 8000-180	Linpack 100 × 100	234	156	0.67	[3]
HP 8000-180	BT	118	59	0.50	[5]
HP 8000-180	BT	34	59	1.84	[5], assuming poor use of prefetching.
HP 8000-180	LU	108	65	0.61	[5]
HP 8000-180	SP	77	56	0.73	[5]
HP 8500-440	Linpack 100 × 100	467	375	0.80	[3]
HP 8500-440	F3D-shared	507	311	0.61	
HP 8500-440	F3D-shared	417	311	0.75	Assuming prefetching is 40% effective.
HP 8500-440	F3D-shared	380	311	0.82	Assuming prefetching is 30% effective.
HP Superdome	F3D-shared	671	527	0.79	
HP Superdome	F3D-shared	580	527	0.91	Assuming prefetching is 40% effective.
HP Superdome	F3D-shared	558	527	0.94	Assuming prefetching is 30% effective.

Notes:

- The NAS benchmarks BT, CG, LU, and SP are for the class B data set using the MPI version of the code (NPB 2).
- Some codes do not seem to lend themselves to benefiting from certain architectural features (e.g., prefetching and multiply-add [madd] instructions). In some cases, a second set of predicted values was calculated to see if turning off a particular feature would bring the prediction more in line with the measured value.
- There has also been some confusion caused by multiple versions of the MPI NAS benchmarks. Over time, the optimization of these codes has improved (at least for some platforms). Depending on which version of code was used to collect the Perfex data and which version was used for the measured data, the results can vary by more than what would be expected.
- In estimating the working sizes of potential working sets, we had to make some guesses. Working sets that fit in the L1 cache are known to be no more than 32 kB in size (the size of the L1 cache on the machines being used to run Perfex). However, we estimate the size of the working set to be 12 kB as a matter of policy.
- Similarly, the size of working sets that fit in the L2 cache, but not the L1, are known to be no more than the size of the L2 (1–8 MB, depending on the system being used). We choose to estimate the size to be 1 MB. However, in some cases, we got significantly better agreement by decreasing the size to what would fit in the cache of a particular machine (e.g., 100 kB for the 128-kB cache of the P2SC or 80 kB for the 96-kB cache of the T3E). The justification for this is that these numbers are more guesses than estimates and therefore should be adjusted to fit the actual data. In most cases where the adjustments were made, the need and appropriateness was immediately apparent since the difference in the predicted level of performance might vary by as much as a factor of 10.

- The available memory bandwidth and memory latency for a shared memory system can be difficult to get right. If the measurements were made using a single processor on a dedicated system, then the measured level of performance might be artificially inflated (the job can use more than its fair share of the memory bandwidth for prefetching. Similarly, if the job can be "locked" onto a single processor of a system with nonuniform memory access times, then the latency might be significantly less than would be otherwise measured (e.g., on the SGI 02K, this can affect performance by up to a factor of 3).
- F3D refers to an implicit CFD code out of NASA Ames that has been modified by the author to run efficiently on shared memory cache-based architectures such as the SGI Origin 2000. A second version of the code was created by Marek Behr, formerly of the U.S. Army High Performance Computing Research Center, and was optimized to run on distributed memory platforms. The shared memory version of F3D uses the native (pre OpenMP) compiler directives to parallelize the code, while the distributed memory version of F3D uses SHMEM calls on systems that support them (e.g., the Cray T3D, Cray T3E, and the SGI Origin 2000) and MPI for other systems. The performance of the shared memory version of the code was measured for single processor runs. The performance of the distributed memory version of the code was measured using eight processor runs, which, in some cases, might result in the underestimation of the single processor performance. Unfortunately, memory limitations make it difficult to run this version on a single processor. The test case was a 1-million grid point projectile (three zones with turbulence turned on).
- CTH is a CSM code out of the DOE running. The test case and results were supplied by Steve Schraml of ARL.

Table 4. A sample run of the program that uses Perfex data to suggest the input parameters for use with the program ENVELOPE.

```
$ envelope.perfex-guide
This program is designed to request a limited amount of information (some hardware and some from
running PERFEX or a similar tool) and then to output a recommended set of input for some of the
input values requested by the program envelope. This program makes heavy use of heuristics, so in
no way is it as accurate as a line-by-line analysis of the source code. However, in many cases, it will be
good enough. One point of caution: The values for "the scratch array memory footprint" and "the
block size" are guesses. They could be larger than these guesses (up to the limits of the size of the L1
and L2 cache, respectively). It is even possible that work assigned to large global blocked arrays
represents a second working set that should be assigned to the scratch arrays or vice-versa. The
rationale for doing things the way they have been done is that it supports two distinct working set
sizes within the constraints of the ENVELOPE program.
============================================================
We will start off by trying to estimate the number of floating point MADD, ADD, and MULTIPLY
instructions. This is an imperfect process. In particular, it is sometimes difficult to know what to call a
MADD, since the SGI hardware can efficiently process independent ADDS and MULTIPLIES in the
same cycle. In theory, this can result in up to a factor of 2 difference between the predicted and
measured levels of performance. The only solution to this problem is to compare the prediction for the
system used to run PERFEX, with the measured number, and to then fine-tune the numbers
accordingly.

This section of the program can work in three ways:
1) Skip this section entirely.
2) Combine Perfex data with an a priori knowledge of the total number of floating point operations to
estimate things.
NOTE: The a priori knowledge can be easily gained by measuring the number of floating point
instructions with MADDS turned off. On the SGI Origin, this is done by compiling with the -mips3
option.
3) Combine Perfex data with numbers from the Cray Hardware Performance Monitor to estimate
things.

What do you want to do (enter 1, 2, or 3)?
2
What is the total number of floating point operations?
5.8937688E10
What is the number of Graduated Floating Point instructions (from Perfex)?
27917089584
For the purpose of running ENVELOPE, it is estimated that there are:
2.9468844E+10 the number of madds
0.0000000E+00 the number multiplies, etc.
0.0000000E+00 the number of adds, etc.
NOTE: Given the input, it is generally impossible to precisely know the ratio between ADD and
MULTIPLY instructions, but for the purpose of this program, it doesn't matter.

Unless you know the memory footprint (e.g., use the number from TOP for RSS), you might want to
assume 1024 MB.

Unless you know the size of the group, assume 1.

Unless you know the density of the group, assume 1.

Unless the code does a lot of integer operations, other than for address calculation, assume 0.0.
============================================================
```

Table 4. A sample run of the program that uses Perfex data to suggest the input parameters for use with the program ENVELOPE (continued).

```
What is the line size for the L1 cache in bytes (32 bytes on the O2K)?
32
What is the line size for the L2 cache in bytes (128 bytes on the O2K)?
128
What is the size of a page of memory (for an Origin 2000 or Origin 3000, use 16 kB) in kB?
16
What is the size of the data item in bytes (usually 8)?
8
How many LOADS graduated (from Perfex)?
84240901392
How many STORES graduated (from Perfex)?
1128365872
What is the L1 Miss rate (from Perfex)?
36720735_ _3552
What is the L2 Miss rate (from Perfex)?
2598239936
What is the TLB Miss rate (from Perfex)?
22215984
-----------------------------------------
Additional values to use as input for ENVELOPE are as follows.
128.0000  cache line size in bytes
16.00000  page size in kB
627.6436  the amount of data being loaded into the
      proc. In GB
8.406981  the amount of data being stored by the
      proc. In GB

 1.000000     Used RLSVDEDICATED
0.0000000E+00 Percentage LSVDEDICATED
 1.000000     Used RSSVDEDICATED
0.0000000E+00 Percentage SSVDEDICATED
0.0000000E+00 Percentage SVMADDS
0.0000000E+00 Percentage SVMULTIPLIES
0.0000000E+00 Percentage SVIOPS
0.0000000E+00 Percentage SSVDEDICATED
0.0000000E+00 Percentage SSVDEDICATED
 1.000000     Used RLSVGENERAL
0.0000000E+00 Percentage LSVGENERAL
 1.000000     Used RSSVGENERAL
0.0000000E+00 Percentage SSVGENERAL
 1.000000     Used RLSADEDICATED
0.0000000E+00 Percentage LSADEDICATED
 1.000000     Used RSSADEDICATED
0.0000000E+00 Percentage SADEDICATED
0.0000000E+00 Percentage SAMADDS
0.0000000E+00 Percentage SAMULTIPLIES
0.0000000E+00 Percentage SAADDS
0.0000000E+00 Percentage SAIOPS
 1.000000     Used RLSAGENERAL
0.0000000E+00 Percentage LSAGENERAL
 1.000000     Used RSSAGENERAL
0.0000000E+00 Percentage SSAGENERAL
1.1700000E-02 the scratch array memory footprint
 1.000000      Used CLGB
0.0000000E+00   Percentage LGB
 1.000000      Used CSGB
0.0000000E+00   Percentage SGB
```

Table 4. A sample run of the program that uses Perfex data to suggest the input parameters for use with the program ENVELOPE (continued).

0.0000000E+00	Percentage GBMADDS
0.0000000E+00	Percentage GBMULTIPLIES
0.0000000E+00	Percentage GBADDS
0.0000000E+00	Percentage GBIOPS
-1.000000	the block size
1.000000	Used CLGN
2.2634468E-03	Percentage LGN
1.000000	Used CSGN
2.2634468E-03	Percentage SGN
2.2634468E-03	Percentage GNMADDS
2.2634468E-03	Percentage GNMULTIPLIES
2.2634468E-03	Percentage GNADDS
2.2634468E-03	Percentage GNIOPS
2.055018	Used CLG1
99.99773	Percentage LG1
2.055018	Used CSG1
99.99773	Percentage SG1
99.99773	Percentage G1MADDS
99.99773	Percentage G1MULTIPLIES
99.99773	Percentage G1ADDS
99.99773	Percentage G1IOPS

8. References

1. Mashey, J. Comp.arch Newsgroup. 25 October 2000.
2. Manjikian, N. "Multiprocessor Enhancements of the Simple Scalar Tool Set." *Computer Architecture News*, vol. 29, no. 1, New York: ACM Press, March 2001.
3. The results for the Linpack (100 × 100) Benchmark maintained as part of the Performance Database Server at <http://www.netlib.org>.
4. Levesque, J. Personal communication. IBM Research, 12 December 2000.
5. "Complete NPB 2 Data 11/17/97: Graphs and Tables." Published electronically at <http://www.nas.nasa.gov/Software/NPB>.
6. Hisley, D., C. Zoltani, and P. Satya-narayana. Personal communication. U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, and Raytheon, U.S. Army Research Laboratory–Major Shared Resource Center, Aberdeen Proving Ground, MD, 2000.
7. Behr, M. Personal communication. U.S. Army High Performance Computing Research Center, Aberdeen Proving Ground, MD, 2000.
8. Schraml, S., and K. Kimsey. Personal communication. U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 2001.
9. Performance results published electronically at <http://www.ncsa.org>.
10. "SGI Origin 3000 Series Performance Report 1.0." Published electronically at <http://www.sgi.com>. 20 September 2000.
11. "IBM Redbook SG24-5155-00." Published electronically at <http://www.ibm.com>. October 1998.
12. "IBM Redbook SG24-5611-00." Published electronically at <http://www.ibm.com>. 23 December 1999.
13. Cappello, F., and D. Etiemble. "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks." Published in the conference proceedings for SC2000 and electronically at <http://www.sc2000.org>.
14. Performance results published electronically at <http://www.ibm.com>.
15. Snavely, A. Personal communication. San Diego Supercomputer Center, University of California at San Diego, CA, 2001.
16. Performance results published electronically at <http://www.nersc>.

17. Patel, J. "ParkBench and EuroBen Benchmarks on the AlphaServerSC." Published electronically at <http://www.cs.utk.edu/~patel/paper html>.

18. Saarinen, S. Personal communication. National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, IL, 2001.

19. Hsieh, J., T. Leng, V. Mashayekhi, and R. Rooholamini. "Architectural and Performance Evaluation of GigaNet and Myrinet Interconnects on Clusters of Small-Scale SMP Servers." Published in the conference proceedings for SC2000 and electronically at <http://www.sc2000.org>.

Glossary

ARL	U.S. Army Research Laboratory
BLAS	Basic Linear Algebra Subprograms
CFD	Computational Fluid Dynamics
CISC	Complicated Instruction Set Computer – an approach to processor design that assumes that the best way to get good performance out of a system is to provide instructions that are designed to implement key constructs (e.g., loops) from **high-level languages**
CSM	Computational Structural Mechanics
CPU	Central Processing Unit
GFLOPS	Billion Floating Point Operations per Second
High-Level Languages	Computer languages that are designed to be relatively easy for the programmer to read and write. Examples of this type of language are FORTRAN, COBOL, C, etc.
kB	Thousand Bytes
Low-Level Languages	Computer languages that are designed to reflect the actual instruction set of a particular computer. In general, the lowest level language is known as Machine Code. Just slightly above Machine Code is a family of languages collectively known as Assembly Code.
MB	Million Bytes
MFLOPS	Million Floating Point Operations per Second
MHz	Million Hertz (cycles/second)
MPI	Message-Passing Interface
MSRC	Major Shared Resource Center
NAS	Numerical Aerospace Simulation – a division of the Information Sciences and Technology Directorate at NASA Ames Research Center, Moffett Field, CA
PAPI	Performance Application Programming Interface

RISC	Reduced Instruction Set Computer - an approach to processor design that argues that the best way to get good performance out of a system is to eliminate the Micro Code that CISC systems use to implement most of their instructions. Instead, all of the instructions will be directly implemented in hardware. This places obvious limits on the complexity of the instruction set, which is why the complexity had to be *reduced*.
SMP	Symmetric Multiprocessor
SPEC	Standard Performance Evaluation Corporation - a company formed to create industry standard benchmarks (mostly for desktop systems)

NO. OF COPIES	ORGANIZATION
2	DEFENSE TECHNICAL INFORMATION CENTER DTIC OCA 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	HQDA DAMO FDT 400 ARMY PENTAGON WASHINGTON DC 20310-0460
1	OSD OUSD(A&T)/ODDR&E(R) DR R J TREW 3800 DEFENSE PENTAGON WASHINGTON DC 20301-3800
1	COMMANDING GENERAL US ARMY MATERIEL CMD AMCRDA TF 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	INST FOR ADVNCD TCHNLGY THE UNIV OF TEXAS AT AUSTIN 3925 W BRAKER LN STE 400 AUSTIN TX 78759-5316
1	US MILITARY ACADEMY MATH SCI CTR EXCELLENCE MADN MATH THAYER HALL WEST POINT NY 10996-1786
1	DIRECTOR US ARMY RESEARCH LAB AMSRL D DR D SMITH 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRL CI AI R 2800 POWDER MILL RD ADELPHI MD 20783-1197

NO. OF COPIES	ORGANIZATION
3	DIRECTOR US ARMY RESEARCH LAB AMSRL CI LL 2800 POWDER MILL RD ADELPHI MD 20783-1197
3	DIRECTOR US ARMY RESEARCH LAB AMSRL CI IS T 2800 POWDER MILL RD ADELPHI MD 20783-1197
	ABERDEEN PROVING GROUND
2	DIR USARL AMSRL CI LP (BLDG 305)

NO. OF COPIES	ORGANIZATION
1	HPCMO C HENRY PRGM DIR 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	HPCMO L DAVIS DPTY PRGM DIR 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	HPCMO V THOMAS DISTRIB CTRS PRJT OFCR 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	HPCMO J BAIRD HPC CTRS PRJT MGR 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	HPCMO L PERKINS CHSSI PRJT MGR 1010 N GLEBE RD STE 510 ARLINGTON VA 22201
1	RICE UNIVERSITY M BEHR MECHL ENGNRG MTRLS SCI 6100 MAIN ST MS 321 HOUSTON TX 77005
1	RICE UNIVERSITY T TEZDUYAR MECL ENGRG MTRLS SCI 6100 MAIN ST MS 321 HOUSTON TX 77005
1	J OSBURN CODE 5594 4555 OVERLOOK RD BLDG A49 RM 15 WASHINGTON DC 20375-5340

NO. OF COPIES	ORGANIZATION
1	NAVAL RSRCH LAB J BORIS CODE 6400 4555 OVERLOOK AVE SW WASHINGTON DC 20375-5344
1	NAVAL RSRCH LAB D PAPACONSTANTOPOULOS CODE 6390 WASHINGTON DC 20375-5000
1	NAVAL RSRCH LAB G HEBURN RSRCH OCNGRPHR CNMOC BLDG 1020 RM 178 STENNIS SPACE CTR MS 39529
1	AIR FORCE RSRCH LAB DEHE R PETERKIN 3550 ABERDEEN AVE SE KIRTLAND AFB NM 87117-5776
1	AIR FORCE RSRCH LAB INFO DIRCTRT R W LINDERMAN 26 ELECTRONIC PKWY ROME NY 13441-4514
1	R A WASILAUSKY SPAWARSYSCEN D4402 BLDG 33 RM 0071A 53560 HULL ST SAN DIEGO CA 92152-5001
1	USAE WTRWYS EXPRMNT STA CEWES HV C J P HOLLAND 3909 HALLS FERRY RD VICKSBURG MS 39180-6199
1	USA CECOM RDEC AMSEL RD C2 B S PERLMAN FT MONMOUTH NJ 07703
1	SPACE AND NVL WRFR SYS CTR K BROMLEY CODE D7305 BLDG 606 RM 325 53140 SYSTEMS ST SAN DIEGO CA 92152-5001

NO. OF COPIES	ORGANIZATION
3	USA HPCRC B BRYAN P MUZIO V KUMAR 1200 WASHINGTON AVE S MINNEAPOLIS MN 55415
1	USA HPCRC G V CANDLER 1200 WASHINGTON AVE S MINNEAPOLIS MN 55415
1	NCCOSC L PARNELL NCCOSC RDTE DIV D3603 49590 LASSING RD SAN DIEGO CA 92152-6148
1	UNIVERSITY OF TENNESSEE S MOORE INNOVATIVE COMPUTER LAB 1122 VOLUNTEER BLVD STE 203 KNOXVILLE TN 37996-3450
1	SDSC UNIV OF CA SAN DIEGO A SNAVELY 9500 GILMAN DR LA JOLLA CA 92093-0505
1	NCSA 152 CAB S SAARINEN 605 E SPRINGFIELD AVE CHAMPAIGN IL 61820
1	USA ERDC D DUFFY CMPTTNL MGRTN GRP MAJOR SHARED RESRC CTR VICKSBURG MS 39180
1	USA ERDC J HENSLEY CMPTTNL MGRTN GRP MAJOR SHARED RESRC CTR VICKSBURG MS 39180
1	USA ERDC M FAHEY CMPTTNL MGRTN GRP MAJOR SHARED RESRC CTR VICKSBURG MS 39180
1	USA ERDC T OPPE CMPTTNL MGRTN GRP MAJOR SHARED RESRC CTR VICKSBURG MS 39180
1	USA ERDC W WARD CMPTTNL MGRTN GRP MAJOR SHARED RESRC CTR VICKSBURG MS 39180
1	USA ERDC R ALTER CMPTTNL MGRTN GRP MAJOR SHARED RESRC CTR VICKSBURG MS 39180
	ABERDEEN PROVING GROUND
20	DIR USARL AMSRL CI N RADHAKRISHNAN AMSRL CI H C NIETUBICZ S THOMPSON AMSRL CI HC P CHUNG J CLARKE D GROVE D HISLEY M HURLEY A MARK D PRESSEL R NAMBURU D SHIRES R VALISETTY C ZOLTANI AMSRL CI HI A PRESSLEY AMSRL CI HS D BROWN T KENDALL P MATTHEWS K SMITH R PRABHAKARAN

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 2002	Final, 1 October 2000 – 15 August 2001

4. TITLE AND SUBTITLE
ENVELOPE: A New Approach to Estimating the Delivered Performance of High Performance Processors

5. FUNDING NUMBERS
665803.731

6. AUTHOR(S)
Daniel M. Pressel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-CI-HC
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-2671

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT*(Maximum 200 words)*

Simulating a computer run can be an excellent method for identifying performance bottlenecks and is especially valuable when discussing systems that do not yet exist. Traditional simulations collect a program trace and then have a simulator execute some subset of the trace one instruction at a time. Unfortunately, all of the standard variants of this technique are far too slow to use on jobs for high-end High Performance Computers and Supercomputers. We have developed an approach based primarily on an analysis of the memory access patterns and the number of floating point operations being executed that will estimate the performance of any run in a small fixed amount of time (e.g., a few seconds or less). Experience has shown that the results are nearly always within a factor of 2 of the measured results and frequently are within 15% or better of the measured results.

14. SUBJECT TERMS
high performance computing, supercomputer, performance modeling

15. NUMBER OF PAGES
54

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.